Enumerating the Strings of a Regular Expression

Panagiotis Manolios

December 5, 2000

1 Introduction

We present a Haskell solution to the problem of enumerating the strings of a regular expression with respect to a regular preorder, a term we soon define. A version of this problem was considered in a note by Jayadev Misra [3]. We have generalized the problem. The use of Haskell was suggested by Edsger W. Dijkstra and our solution makes critical use of Haskell's lazy evaluation.

1.1 Regular Expressions

A is an *alphabet*, a set of symbols. A^* denotes the set of *strings*, finite sequences over A . The empty string is denoted by ϵ and $\cdot$ denotes the concatenation operator on A^* .

A regular expression over A takes one of the following six forms.

1. $\emptyset$
2. ϵ
3. An element of A
4. $p \mid q$
5. $p \bullet q$
6. p^*

where p and q are regular expressions. $L.r$, a subset of A^* , is the *language* denoted by regular expression r . $L.\emptyset$ is the empty set. If $a \in A$ or $a = \epsilon$ then $L.a = \{a\}$.[1] $L.(p \mid q) = L.p \cup L.q$. $L.(p \bullet q) = L.p \times L.q$, where $B \times C = \{b \cdot c : b \in B \wedge c \in C\}$. $L.p^* = \cup_{i \in \mathbb{N}} p^i$, where $p^0 = \{\epsilon\}$ and $p^{i+1} = p \times p^i$. There are many good books that discuss the theory of regular expressions [1]. Note that in the standard treatment of regular expressions, the alphabet is required to be finite, whereas we have no such restriction.

[1] We denote a single element sequence and its element in the same way.

1.2 Regular Preorders

Henceforth, u, v, w, x, y, z range over A^*. Let $\preceq$ be a *regular preorder*. By this we mean that for all w, x, y, z all of the following hold.

1. $x \preceq y \quad \vee \quad y \preceq x$
2. $x \preceq y \quad \wedge \quad y \preceq z \quad \Rightarrow \quad x \preceq z$
3. $\preceq$ is computable
4. $\epsilon \preceq x$
5. $x \preceq y \quad \wedge \quad w \preceq z \quad \Rightarrow \quad x \cdot w \ \preceq \ y \cdot z$

The first property is *totality* and implies *reflexivity*; the second is *transitivity*. A *preorder* is a reflexive, transitive relation. From the first two properties we have that $\preceq$ is a total preorder. Since we construct an algorithm we require, with the third condition, that $\preceq$ is computable. The final two conditions, that ϵ is bottom and that $\cdot$ is monotonic with respect to $\preceq$, can be used to show that regular languages are well-founded with respect to $\preceq$, as we will see later.

1.3 The Enumeration Problem

We abbreviate $x \preceq y \ \wedge \ \neg(y \preceq x)$ by $x \prec y$. We use $\#$ to denote both the length of a list and the cardinality of a set. A list l is an *enumeration* of a regular expression r with respect to regular preorder $\preceq$ iff all of the following hold.

1. Every element of l is in $L.r$.
2. l contains no duplicates.
3. l is *increasing*: for any x and y, if x appears before y in l then $x \preceq y$.
4. l and $L.r$ have the same cardinality, *i.e.*, $\#l \ = \ \#(L.r)$.
5. If $y \in l$, then $\langle \forall x : x \in L.r \ \wedge \ x \prec y : x \in l \rangle$.

There can be many enumerations of r as is the case when $x \preceq y$ for all x, y. But there is at least one enumeration and if $\preceq$ is a total order,[2] then the enumeration is unique. Sometimes no enumeration of r contains all of the strings in $L.r$, *e.g.*, consider $r = a^*|b$ (for $a, b \in A$) where $\langle \forall i \in \mathbb{N} :: a^i \prec b \rangle$: no enumeration contains b. If all the strings in $L.r$ have a finite set of predecessors in $L.r$, then any enumeration contains all of the strings in $L.r$. The enumeration problem is to find an algorithm that, given regular expression r and regular preorder $\preceq$, returns an enumeration of r with respect to $\preceq$.

[2] A *total order* (sometimes referred to as a linear order) satisfies the first two conditions of a regular preorder and antisymmetry: $x \preceq y \ \wedge \ y \preceq x \ \Rightarrow \ x = y$.

2 A Haskell Solution

Our solution to the enumeration problem is written in Haskell, a programming language that is functional, lazy, and allows infinite objects. Function application is invisible; functions are defined using pattern matching; `[]` denotes the empty list; `a:p` is the list whose head is `a` and whose tail is `p`; `tail p` is the tail of `p`; and `++` is the concatenation operator on lists and strings, which we represent as lists in Haskell. We will comment on other notational conventions as the need arises.

We define the function `enum` to solve the enumeration problem. The definition of `enum` is mutually recursive. Functions in the mutually recursive nest can be used on infinite lists and thus can be used to compute any finite prefix of an enumeration. The first three forms are easy; we focus on the remaining three. We give an informal proof of correctness by induction on the structure of regular expressions. The full code appears in the appendix.

2.1 Union

We exhibit `enumUnion`, a function that computes the enumeration of $p \mid q$ with respect to the regular preorder `<<=`. The idea is to merge the enumerations of `p` and `q` and to remove duplicates.

```
enumUnion p q       = removeDups (merge (enum p) (enum q))

removeDups []       = []
removeDups (x0:xs) = x0 : removeDups [z|z<-xs, z /= x0]

merge [] y          = y
merge x []          = x
merge (x0:xs) (y0:ys)
  | x0 <<= y0       = x0 : merge xs (y0:ys)
  | otherwise       = y0 : merge (x0:xs) ys
```

The function `removeDups` is defined using pattern matching. The patterns are checked from top to bottom until a matching pattern is found. The second pattern defining `removeDups` is matched if its argument is non-empty; in this case the argument is a list whose first element is `x0` and whose tail is `xs`. We also see an example of list comprehension: `removeDups` is called recursively on the list consisting of the elements in the tail of its argument that differ from the head. The argument to `removeDups` can be an infinite list, in which case the function never terminates but any finite prefix can be computed.

The function `removeDups` removes duplicates in a list while maintaining the order of the remaining elements. Thus, if the argument to `removeDups` is increasing, so is the result. The function `merge` merges its arguments and if the arguments are increasing, so is the result.

The proof of this case follows. The induction hypothesis (IH) consists of the five parts IH1, ..., IH5 of the definition of enumeration and states that `(enum p)` and `(enum q)` are enumerations of `p` and `q`, respectively. We now show that `(enumUnion p q)` is an enumeration of `p | q`; this consists of checking the five conditions comprising the definition of enumeration.

1. Every element of `enumUnion p q` is an element of `merge (enum p) (enum q)` which is either an element of `enum p` or `enum q` and thus, by IH1, an element of $L.$(p | q).
2. `enumUnion` has no duplicates because it is defined with `removeDups`.
3. By IH3, `enum p` and `enum q` are increasing, therefore `merge (enum p) (enum q)` is increasing, thus `enumUnion p q` is increasing.
4. If $L.$(p | q) is infinite then $L.$p or $L.$q is infinite; by IH4 either `enum p` or `enum q` is infinite and thus so is `enumUnion p q`. Otherwise, both $L.$p and $L.$q are finite and from IH1, IH2, IH4 we have that `enum p` and `enum q` contain exactly the strings in $L.$p and $L.$q, respectively. By `merge`, `enumUnion p q` contains exactly the strings in $L.$(p | q) and by 2, above, `enumUnion p q` contains no duplicates. Thus, `#(enumUnion p q)` $=$ $\#L.$(p | q).
5. Let `y` be in `enumUnion p q`; let `x` be in $L.$(p | q) and `x << y`.[3] Since `y` is in `enumUnion p q`, it is in `enum p` or `enum q`. Suppose it is in `enum p`. If `x` $\in L.$ `p` then by IH5 it is in `enum p` and by IH3 it appears before `y`, thus by `merge`, it appears in `enumUnion p q`. If `x` $\in L.$ `q` then by `merge` and IH3, we have that every element in `enum q` which is `<< y` appears in `enumUnion p q` and our proof obligation is to show that `x` is in `enum q`. If `y` appears after all the elements of `enum q` then by 4, above, `enum q` contains all of the elements in $L.$q and thus also `x`. Otherwise, `merge` has reached an element `z` in `enum q` such that `x << z`, thus by IH5, `x` is in `enum q` and by `merge` in `enumUnion p q`. If `y` is in `enum q` the proof is similar.

2.2 Concatenation

We define `enumConcat` to compute the enumeration of $p \bullet q$. We create a matrix of strings with as many rows as there are strings in the enumeration of p and as many columns as there are strings in the enumeration of q. The string in a cell of the matrix is obtained by concatenating the string associated with the row of the cell and the string associated with the column of the cell. Using the monotonicity of `++` with respect to `<<=`, we have that rows and columns of the matrix are increasing, thus we can enumerate the elements in the matrix by replacing the first two rows by their merge and recurring.

```
enumConcat p q   = removeDups (multiMerge (enum p) (enum q))
```

[3] `<<` is a Haskell version of $\prec$.

```
multiMerge [] _  = []
multiMerge _  [] = []
multiMerge ep eq = matrixMerge (map makeRow ep)
                 where makeRow p = map (p++) eq

matrixMerge [r]         = r
matrixMerge (r0:r1:mx) = a : matrixMerge (r01:mx)
                        where (a:r01) = merge r0 r1
```

Above we use the function `map`, a function whose first argument is a function, whose second argument is a list, and whose value is the list obtained by applying its first argument to every element of its second argument. We also use the *section* `(p++)` : this is a function that prepends `p` to its argument.

We define the matrix in the third pattern of `multiMerge` : it is the argument to `matrixMerge` and is represented as a list of rows. Function `matrixMerge` returns the first row of its argument, if it has only one row. Otherwise, denote the first row `r0` , the second `r1` , and what remains `mx` . We merge `r0` and `r1` with the head of the result denoted `a` and the tail `r01` ; the list returned contains `a` as the head while the tail is the `matrixMerge` of `r01:mx` . We break up the merge of the first two rows into `a` and `r01` so that, with the use of lazy evaluation, any finite prefix of the result can be computed. We make use of the following lemma.

Lemma 1 *M is a matrix of strings whose rows and columns are associated with duplicate-free, increasing (with respect to $\preceq$) strings and whose elements are obtained by concatenating the string corresponding to the element's row with the string corresponding to the element's column. For any $i \in \mathbb{N}$ not exceeding the maximum of the number of rows and columns in M , consider l , an ordered list containing the strings (sans duplicates) in the grid, with the constraint that for any string in the grid, all strings in the same row, but smaller column appear before it in l and all strings in the same column, but smaller row appear before it in l . Then the first i strings in l are $\preceq$ all other strings in M .*

Proof Since the rows and columns have no duplicates, there are at least i distinct strings within the $i \times i$ grid. Since $\preceq$ is a regular preorder, we can order the strings (sans duplicates) in the grid as required. By the monotonicity of `++` over `<<=` , for any element outside the grid there are at least i strings in the grid that are $\preceq$ to it, hence, the first i elements are `<<=` the element. Finally, the elements in the grid are ordered, by construction. $\Box$

The proof of the concatenation case follows. By IH, `enum p` and `enum q` are enumerations of `p` and `q` , respectively. We now show that `enumConcat p q` is an enumeration of $\texttt{p} \bullet \texttt{q}$. The case where $L.\texttt{p}$ or $L.\texttt{q}$ is empty is easy, so we assume that the languages are not empty.

1. The strings of `enumConcat p q` are from the matrix `map makeRow ep` and thus are obtained by concatenating strings in `enum p` with strings in `enum q` and thus are strings of $L.(\texttt{p} \bullet \texttt{q})$, by IH1.

2. `enumConcat` has no duplicates because it is defined with `removeDups`.

3. `enumConcat p q` is increasing: we use Lemma 1 and note that the i^{th} element is selected from the first i rows of the matrix.

4. If $L.(\mathtt{p} \bullet \mathtt{q})$ is infinite then $L.\mathtt{p}$ or $L.\mathtt{q}$ is infinite and so is the matrix. By Lemma 1, the matrix has an infinite number of distinct strings and thus so does `enumConcat p q`. Otherwise, both $L.\mathtt{p}$ and $L.\mathtt{q}$ are finite and from the IH1, IH2, IH4 we have that `enum p` and `enum q` contain exactly the strings in $L.\mathtt{p}$ and $L.\mathtt{q}$, respectively. By construction, the matrix contains exactly the strings in $L.(\mathtt{p} \bullet \mathtt{q})$ and by 2, above, `enumConcat` contains no duplicates, thus `#(enumConcat p q)` $=$ $\#L.(\mathtt{p} \bullet \mathtt{q})$.

5. Let `y` be in `enumConcat p q`; let `x` be in $L.(\mathtt{p} \bullet \mathtt{q})$ and `x << y`. By Lemma 1, every element in the matrix `<< y` appears before `y`. What is left is to show that `x` appears in the matrix. If it does not, then `x` $=$ `a ++ b` where `a` is in $L.\mathtt{p}$, `b` is in $L.\mathtt{q}$, and `a` is not in `enum p` or `b` is not in `enum q`. Say that `a` is not in `enum p`; then `enum p` is infinite and every element in `enum p` is `<<= a`, by IH5. Thus, by the monotonicity of `++` with respect to `<<=`, every element in a column corresponding to a string `<<= b` is `<< y`, but there are an infinite number of elements in the column, hence, we have a contradiction. A similar argument can be used if `b` is not in `enum q`.

2.3 Kleene closure

The function `enumStar`, defined below, computes the enumeration of p^*.

```
enumStar p
  | epNoE == []     = [[]]
  | otherwise       = ps
  where epNoE       = [z|z <- (enum p), z /= []]
        ps          = removeDups ([] : (multiMerge epNoE ps))
```

In the above definition, `epNoE` is `enum p` with `[]`, the empty string in Haskell, removed. If `epNoE` is empty, $L.\mathtt{p}^*$ contains only the empty string. Otherwise, $L.\mathtt{p}^*$ is infinite and is the least fixpoint of the following equation in X : $X = \{\mathtt{[]}\} \cup L.\mathtt{p} \times X$. Notice that `ps` is defined to satisfy the above equation.[4] As an example of lazy evaluation in Haskell, note that the second element of `ps` is the element in the first row and first column of the matrix defined by `multiMerge epNoE ps` which is the first element of `epNoE`, as the first element of `ps` is `[]`. The second element can be used to obtain the third element and so on as this is a non-terminating process.

The proof of this case follows. We already examined the case where $L.\mathtt{p}^* = \{\ \mathtt{[]}\ \}$, so we assume that $L.\mathtt{p}$ contains a string that differs from `[]`. By IH, `enum p` enumerates `p`. We now show that `ps`, which equals `enumStar p`, is an enumeration of $\mathtt{p}^*$.

[4] We add the minor optimization of using `epNoE` instead of `enum p`.

1. The elements of `ps` include `[]` and are otherwise obtained by repeated concatenation of strings in `enum p`, which, by IH1, are strings in $L.\mathtt{p}$, thus, the elements of `ps` are strings in $L.\mathtt{p}^*$.

2. `ps` has no duplicates because it is defined with `removeDups`.

3. We show by induction that for all $i \in \mathbb{N}$, *pref*.i, the prefix of `ps` of length i, is well-defined and increasing. When $i = 2$ *pref*.i is `[[], a]` where `a` is the first element in `epNoE` (recall that `a` $\neq$ `[]`).[5] For $i \geq 2$ the $(i+1)^{th}$ element of `ps` is selected from the $i \times i$ grid of the matrix whose rows correspond to `epNoE` and whose columns to *pref*.i. Since *pref*.i and `epNoE` are increasing and duplicate-free, by Lemma 1, there are i distinct strings in the grid which are `<<=` all other elements in any increasing, duplicate-free extension of the matrix, hence, the i^{th} element can be used to obtain *pref*.$(i+1)$.

4. In 3, above, we showed that *pref*.i is well-defined for all natural numbers, hence, `ps` is infinite, as is $L.\mathtt{p}^*$.

5. Let `y` be in `ps`; let `x` be in $L.\mathtt{p}^*$ and `x << y`. By Lemma 1, every element in the matrix `<< y` appears before `y`. What is left is to show that `x` appears in the matrix. If it does not, then let `z` be the shortest suffix of `x` in $L.\mathtt{p}^*$ that does not appear in `ps`. We have `z` = `a ++ b` where `a` is in $L.\mathtt{p} \setminus \{$ `[]` $\}$, `b` is in $L.\mathtt{p}^*$, and `a` is not in `epNoE` or `b` is not in `ps`. Since `a` $\neq$ `[]`, `b` is in `ps` by the minimality of `z`, hence, `a` is not in `epNoE`. Hence, by IH1, IH2, IH4, `epNoE` is infinite and by IH5 all of its elements are `<<= a`, thus `<<= z`, thus `<<= x`, thus `<< y` and we have a contradiction.

3 Final Remarks

3.1 Subsequence Relation

If x can be obtained by removing elements from y, we say that x is a *subsequence* of y. The following lemma shows that any regular preorder preserves the subsequence relation.

Lemma 2 *If x is a subsequence of y then $x \preceq y$.*

Proof By induction on the length of x.

- Base case ($x = \epsilon$): By ϵ is bottom.

[5] Notice that if we replace `epNoE` with `enum p` in the definition of `ps` and if `[]` is an element of `enum p`, then Haskell will diverge when computing `ps`. Hence, the use of `epNoE` turns out to be important in the presence of lazy evaluation.

- Induction step (x is of the form $w \cdot z$, where z is a sequence of length 1). Since x is a subsequence of y , we have that y is of the form $u \cdot z \cdot v$, where w is a subsequence of u .

$$
\begin{array}{ll}
 & x \\
= & \{ \text{ Form of } x \} \\
 & w \cdot z \\
\preceq & \{ \text{ Induction hypothesis, monotonicity of } \cdot \text{ , reflexivity of } \preceq \} \\
 & u \cdot z \\
\preceq & \{ \text{ Monotonicity of } \cdot \text{ , } \epsilon \preceq v \text{ , reflexivity of } \preceq \} \\
 & u \cdot z \cdot v \\
= & \{ \text{ Form of } y \} \\
 & y
\end{array}
$$

3.2 Total Orders

When we are dealing with a total order we can use the function `compress` instead of `removeDups` .

```
compress []       = []
compress (x:xs)   = x : compress(dropWhile (== x) xs)
```

In the second pattern above we use `dropWhile` , a function that drops the initial sequence of `x`'s from `xs` . We can use `compress` instead of `removeDups` with total orders because both `merge` and `multiMerge` return increasing lists (given increasing lists as input); therefore, duplicate elements are adjacent to one another.

3.3 Example Regular Preorders

The simplest example of a regular preorder is one that relates everything. Its definition in Haskell is:

```
_ <<= _        = True
```

Below is an example of a total regular order with the additional property that all strings have a finite number of predecessors in any regular language, thus, the enumeration of any regular language r with respect to this order is unique and contains all the elements in $L.r$. The Boolean operators $\vee, \wedge$ are denoted `||`, `&&` in Haskell. We do not give a definition for `<=` ; it is the dictionary order induced by a total order on singletons and is built-in for certain Haskell types, *e.g.*, it can be used to order lists of integers and lists of characters.

```
x <<= y
  =  let lenx = length x
         leny = length y
     in (lenx < leny) || ((lenx == leny) && (x <= y))
```

The final example is the component-wise order on the characters `a` and `b`. This regular preorder can be thought of as being of order-type ω^2. By this we mean that there is an order-isomorphism between ω^2 and the set of equivalence classes of the regular order. To see this, note that two strings in $\{$ `a,b` $\}^*$ are in the same class if they are permutations. Finally, the order-preserving bijection from strings to ordinals in w^2 takes string `x` to $\omega\cdot$(number of `a`'s in `x`) $+$ (number of `b`'s in `x`).

```
x <<= y
 = xas < yas || (xas == yas && xbs <= ybs)
   where xas = num 'a' x
         xbs = num 'b' x
         yas = num 'a' y
         ybs = num 'b' y
         num n x = length [z|z<-x, z==n]
```

3.4 Finite-State Automata

Another approach is to turn a regular expression into a minimal deterministic finite-state automaton (DFA) and to enumerate the expression using the DFA. This can be done with a function `enumDfa` whose single argument is a list of pairs, where each pair consists of a string and a state in the DFA. We maintain the invariant that the pairs in the list are sorted with respect to `<<=`, using their strings as keys. In addition, the path through the DFA determined by the string of a pair leads to the state of the pair. The initial call of `enumDfa` consists of the list with the single pair consisting of the empty string and the start state. As long as its argument in not empty, `enumDfa` removes the pair at the head of the list. If it contains an accepting state, the string is added to the enumeration. In either case, the pairs consisting of states reachable in a single step and their corresponding strings are inserted into the list, which is the argument to the recursive call of `enumDfa`.

Acknowledgments

We thank Jayadev Misra for posing an interesting problem and giving a preliminary solution. We thank Edsger W. Dijkstra for suggesting the use of Haskell; the suggestion gave us the incentive to learn Haskell. We presented this work to the ACL2 group and to ATAC and received valuable comments in both cases. John Gunnels read this paper and provided valuable feedback.

Appendix

In the Haskell code appearing below, the text following `--` is a comment and thus is ignored by Haskell. `Reg` is the type of regular expressions. It is a polymorphic type which means that we can build regular expression over any underlying type. The code and various examples are available from our Web site [2].

```
infix 0 <<=             -- <<= is a regular preorder

data Reg a = Nil | E | L a | U (Reg a) (Reg a) |
             C (Reg a) (Reg a) | S (Reg a)
           deriving Show

enum Nil       = []     -- The empty language
enum E         = [[]]   -- The language containing []
enum (L l)     = [[l]]  -- The language containing l
enum (U p q)   = enumUnion p q
enum (C p q)   = enumConcat p q
enum (S p)     = enumStar p

enumUnion p q  = removeDups (merge (enum p) (enum q))

removeDups []      = []
removeDups (x0:xs) = x0:removeDups [z|z<-xs, z /= x0]

merge [] y            = y
merge x []            = x
merge (x0:xs) (y0:ys)
  | x0 <<= y0         = x0 : merge xs (y0:ys)
  | otherwise         = y0 : merge (x0:xs) ys

enumConcat p q = removeDups (multiMerge (enum p) (enum q))

multiMerge [] _  = []
multiMerge _  [] = []
multiMerge ep eq = matrixMerge (map makeRow ep)
                 where makeRow p = map (p++) eq

matrixMerge [r]        = r
matrixMerge (r0:r1:mx) = a:matrixMerge (r01:mx)
                       where (a:r01) = merge r0 r1
enumStar p
  | epNoE == []  = [[]]
  | otherwise    = ps
  where epNoE    = [z|z <- (enum p), z /= []]
        ps       = removeDups ([] : (multiMerge epNoE ps))
```

References

[1] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[2] P. Manolios. Homepage of Panagiotis Manolios, 2000. See URL `http://-www.cs.utexas.edu/users/pete`.

[3] J. Misra. Enumerating the strings of a regular expression, 2000. Unpublished draft.